Zesto on SST

1. Introduction

Zesto is a cycle-level simulator developed for modeling detailed x86 pipelines, including both out-of-order and in-order scheduling and execution that much more closely mirrors current implementation, as well as many x86-specific microarchitecture features. The Zesto project was originated in Georgia Tech by Gabriel Loh and his students. More information on Zesto is available at http://zesto.cc.gatech.edu.

SST is the Structural Simulation Tookit developed in Sandia that enables simulation diverse aspects of hardware and software in co-design of extreme-scale architectures. The toolkit has fully modular structure that enables extensive exploration of an individual system parameter without the need for intrusive changes to the simulator. The MPI-based parallel simulation environment provides a high level of performance and the ability to model large systems. More information on SST is available at http://sst.sandia.gov.

This document describes necessary interfaces that should be established in order to build Zesto as a SST component and setup procedures to connect Zesto in a multi-node network with other uncore components.

2. Zesto on SST

This section describes how Zesto is migrated to SST by declaring new functionality and interfaces in the original Zesto core.

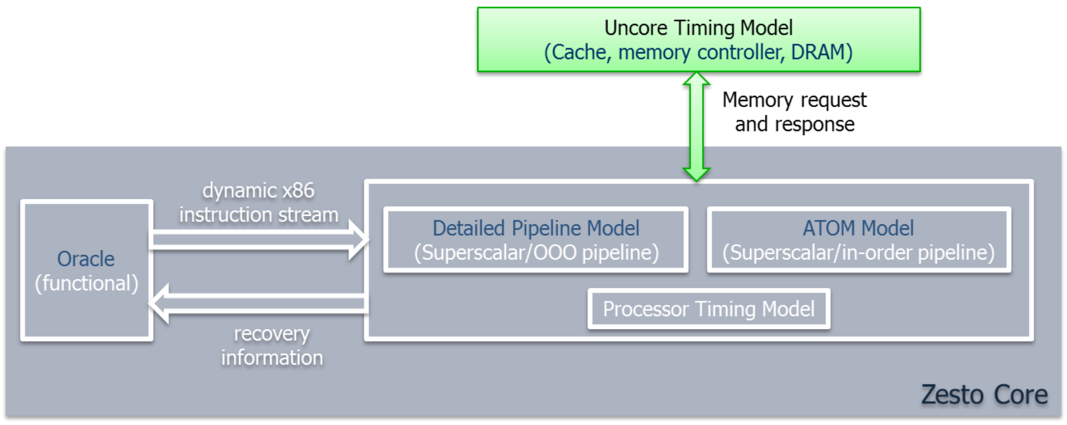
Uncore Timing Model
(Cache, memory controller, DRAM)
Memory request and response
dynamic x86 instruction stream
Oracle (functional)
recovery information
Detailed Pipeline Model (Superscalar/OOO pipeline)
ATOM Model (Superscalar/in-order pipeline)
Processor Timing Model
Zesto Core

Figure 1. Overall Zesto Structure

2.1. Overall Zesto Structure

The overall organization of the Zesto simulator includes a functional oracle simulator and a processor timing mode, as shown in Figure 1. The oracle simulator executes all instruction and provides the dynamic stream of instructions before the processor timing model takes all the stream information and perform detailed pipeline simulation and timing. On a pipeline flush (e.g. due to branch misprediction or a load ordering violation), the processor timing model notifies notifies the oracle which instruction it should roll back to, and the oracle uses this “undo” information to recover the previous state of the pipeline.

Currently Zesto supports two pipeline models/organizations/implementations: a Detailed Pipeline Model (DPM) which is based on modern Intel x86 superscalar out-of-order pipelines and an ATOM model which is based on x86 Atom in-order pipelines. Each pipeline model implements each of the five basic pipeline stages: Fetch, Decode, Allocation, Execute and Commit in a cycle-based manner.

All the memory requests generated by Zesto core are handled by uncore timing models, such as cache, memory controller and DRAM. All these models can be defined to interface with Zesto core properly when modeling multi-processor systems.

Refer to available zesto documents for more detailed implementation of zesto.

2.2. Zesto migration on SST

The overall definition of zesto on SST is shown in Figure 2.

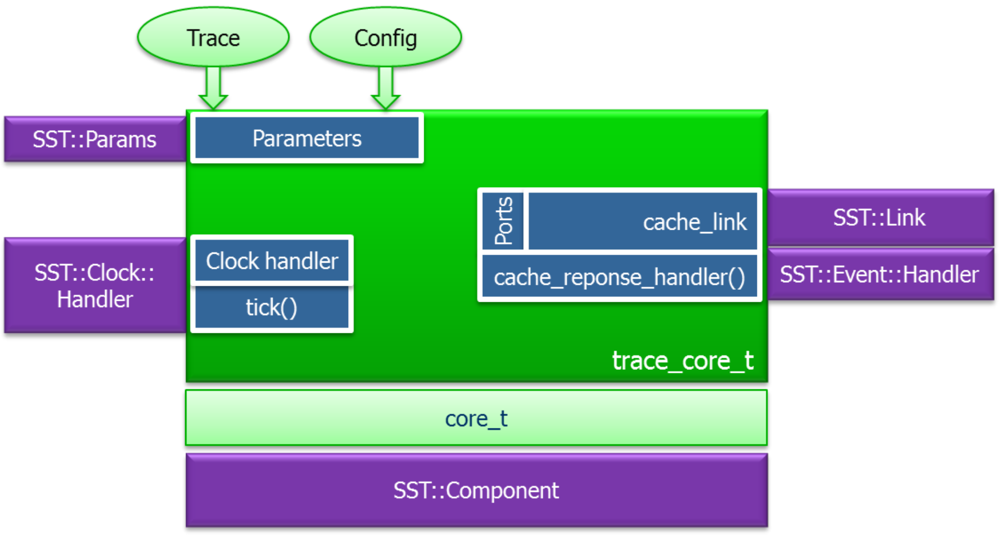

Figure 2. Zesto on SST

Zesto core is defined as a component in SST using the SST Component class. All the functionalities defined in the original Zesto core, including oracle simulation and pipeline modeling, are wrapped in this component implementation.

Zesto is able to take multiple frontends as instruction inputs, including Traces generated by Qsim/PIN, Qsim streams as well as Qsim server packets. Each frontend is defined as a derived class on top of zesto component to maintain a clear interface. In current version of zesto on SST, only trace frontend is supported, with its corresponding class trace_core_t. The implementation of trace_core_t reads in the trace file specified in SST SDL configuration, and invokes zesto core to simulate the trace.

The cycle-by-cycle simulation of Zesto core is performed by the clock handler tick() which is registered to SST in class constructor. When this clock handler is invoked, the timing model of zesto proceeds for one cycle.

Zesto can be connected to an uncore simple cache component through its cache link. The link is configured with cache_response_handler in zesto class constructor by calling link configuration API defined in SST.

Both the zesto configuration file and trace file will be part of SST params to zesto component, along with other parameters such as clock frequency and node id.

2.3. Simple Cache and Simple MC Component

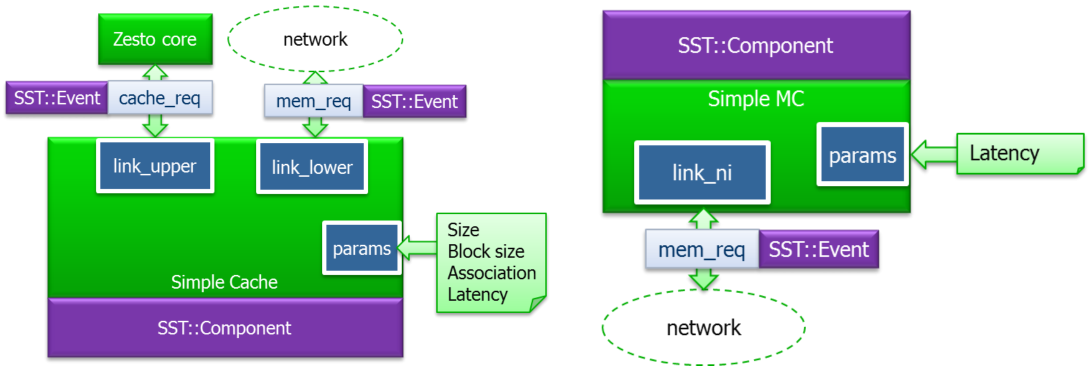

Figure 3. Simple Cache and Simple MC Component

Zesto is connected with a simple cache and a dummy memory controller to form a complete system, as shown in Figure 3.

Simple cache component is defined with two SST links: link_upper and link_lower. The link_upper link is connected to an upward terminal, which in this case is the zesto core. The link_lower link is connected to next-level memory with/without the network. Cache configuration is specified through SST params, including size, block size, association and access latency. Inside the simple cache component, a hash table is maintained to simulate the cache hit/miss behavior.

The dummy memory controller is defined as Simple MC component with one link link_ni connected to either network or simple cache. Simple MC only takes latency as configuration parameter through SST params. Inside simple MC, the received memory requirement will be sent back to network after configured latency time.

The link message type passing between simple cache and zesto core is defined as a SST event cache_req, while the messages routing around the network is defined as a SST event mem_req.

2.4. Iris Network Interface Controller

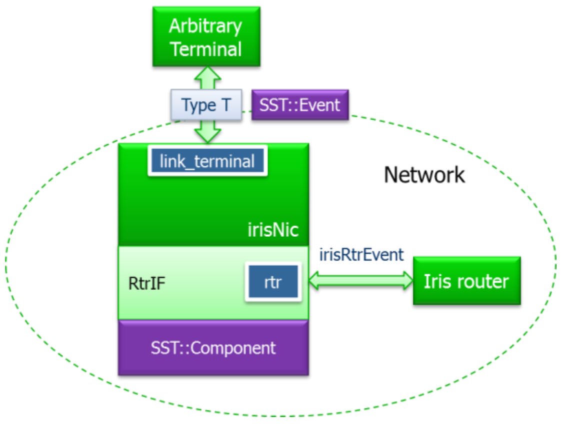

Figure 4. Iris Network Interface Controller

Both simple cache and simple MC can be connected to the Iris router through Iris Network Interface Contoller (NIC) component. Iris router is a developed SST component in SST repository. It is shipped with a general-purpose network interface RtrIF. However, in order to connect simple cache and simple MC to RtrIF, extra functionality is required, which is performed through IrisNIC. IrisNIC can be connected to an arbitrary terminal through its link link_terminal. The message passed on link_terimnal can be an arbitrary type T inherited from SST Event. Event from terminal is first packed into Iris packets format in IrisNIC and then transmit to Iris router through packet sending APIs provided by RtrIF. Similarly, events from Iris router will be received and sent back to IrisNIC through RtrIF. IrisNIC unpacks and reorganizes the packets to send them back to the terminal.

More details of Iris Router and its packet format can be found in SST Iris documentations.

2.5. Typical Multi-Node Network Structure for Zesto

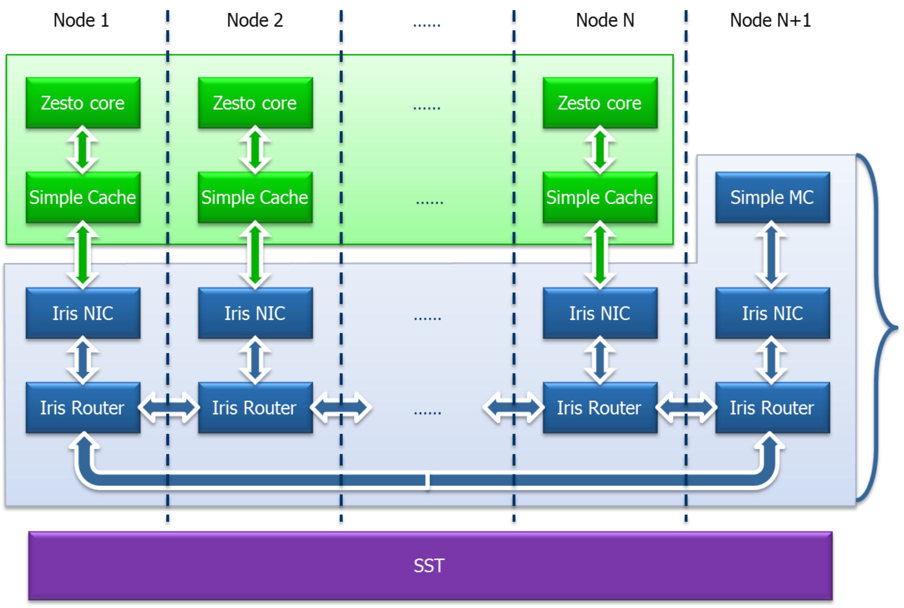

Figure 5. Typical Multi-Node Network

Figure 5 shows a typical N+1 Node network structure with the above SST components: zesto, simple cache, simple MC, Iris NIC and Iris router. Each node contains either a pair of zesto core and simple cache or a simple MC, which is connected to Iris network through Iris NIC. The Iris network can be configured with whatever topological that is supported by Iris router, e.g., mesh, ring, etc. The example in Figure 5 uses ring topology.

3. Running Zesto on SST

3.1. Zesto Directory Structure

Zesto is part of SST that can be found in sst-simulator/sst/elements/zesto. Table 1 gives brief description of each directory.

Directory	Description
zesto	Interface with SST.
zesto/core	Zesto core implementation.
zesto/core_config	Zesto core configuration files.
zesto/doc	Zesto documentation.
zesto/irisNIC	Iris network interface controller.
zesto/sdl	Sample SDL files.
zesto/simpleCache	Simple cache component.
zesto/simpleMC	Simple MC component.
zesto/trace	Sample trace file for zesto.

Table 1. Zesto Directory Structure

3.2. Building Zesto on SST

Zesto is built with SST. Refer to SST documentation for building procedure of SST.

3.3. Configure and Run Zesto

Due to the its current implementation feature on SST, zesto can be configured with or without Iris network. The system is setup by a SDL configuration file.

General format of SDL file can be found in SST documentations. For zesto system, SDL file should include all necessary configuration parameters for each component listed in Table 2.

Component	Parameter	Description
All Components	nodeId	Exclusive node id for each node in the network. All components in a node should have the same nodeId. One node includes a pair of zesto and simple cache, or a simple MC, as well as a Iris NIC and a Iris router (See Figure 5).
Zesto	clockFreq	Clock frequency of the core.
	configFile	Zesto configuration file. Refer to zesto documentation for more details.
	traceFile	Traces generated by Qsim or Pin. Refer to zesto documentation for format of trace file.
simpleCache	size	Cache size.
	assoc	Association.
	block_size	Cache line size.
	hit_time	Cache hit access latency in ns.
	lookup_time	MSHR lookup latency in ns.
	mc_node_ids	Node Id of Simple MC.
simpleMC	clockFreq	Clock frequency of Simple MC.
	latency	Memory access latency.
irisNic	event_type	Event type name. Currently only “mem_req” is supported.
	num_vc	Virtual channel number. Always use 1.
	clock	Clock frequency of IrisNic.
	Node2RouterQSize_flits	Size of the router queue in Nic.
Iris router		Refer to Iris documentations.

Table 2. Parameters for Each SST Component

To configure zesto without network, zesto core is connected to simple cache which is then connected to simple MC directly. In this case, three components are used in one SDL file. The example SDL file for this configuration can be found in zesto/sdl/simple3.xml.

To configure zesto with network, simple cache and simple MC must be connected to Iris router through Iris NIC. In this case, each node in the network may have either four components (zesto + simpleCache + IrisNic + Iris router) or three components (simpleMC + IrisNic + Iris router). Two examples for this configuration can be found in zesto/sdl/simple3_with_nic.xml and zesto/sdl/conf5x1.xml.